

DTIC FILE COPY

APPROVED FOR
PUBLIC DISTRIBUTION

(4)

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

VLSI PUBLICATIONS

VLSI Memo No. 89-571
October 1989

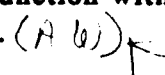
DTIC
ELECTE
JAN 17 1990
S D & D

Irredundant Sequential Machines Via Optimal Logic Synthesis

Srinivas Devadas, Hi-Keung Tony Ma, A. Richard Newton, and Alberto Sangiovanni-Vincentelli

Abstract

It is well known that optimal logic synthesis can ensure fully testable combinational logic designs. In this paper we show that optimal sequential logic synthesis can produce irredundant, fully testable finite state machines. Test generation algorithms can be used to remove all the redundancies in sequential machines resulting in a fully testable design. However, this method may require exorbitant amounts of CPU time. The optimal synthesis procedure presented in this paper represents a more efficient approach to achieve 100% testability.

Synthesizing a sequential circuit from a State Transition Graph description involves the steps of state minimization, state assignment and logic optimization. Previous approaches to producing fully and easily testable sequential circuits have involved the use of extra logic and constraints on state assignment and logic optimization. In this paper we show that *100% testability can be ensured without the addition of extra logic and without constraints on the state assignment and logic optimization*. Unlike previous synthesis approaches to ensuring fully testable machines, there is *no area/performance penalty* associated with this approach. This technique can be used in conjunction with previous approaches to ensure that the synthesized machine is easily testable. (A 6) 

Given a State Transition Graph specification, a logic-level automaton that is fully testable for *all single stuck-at* faults in the combinational logic *without access to the memory elements* is synthesized. This procedure represents an alternative to a Scan Design methodology without the usual area and performance penalty associated with the latter method.

90 01 16 147



Accession For	
NTIS CRA&I	<input checked="checked" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution /	
Availability Codes	
Dist	Avail and/or Special
A-1	

Acknowledgements

This research was supported in part by the Semiconductor Research Corporation, the Defense Advanced Research Projects Agency under contract N00014-87-K-0825, and a grant from AT & T Bell Laboratories.

Author Information

Devadas: Department of Electrical Engineering and Computer Science, Room 36-848, MIT, Cambridge, MA 02139. (617) 253-0454.

Ma, Newton, and Sangiovanni-Vincentelli: Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA 94720.

Copyright© 1989 MIT. Memos in this series are for use inside MIT and are not considered to be published merely by virtue of appearing in this series. This copy is for private circulation only and may not be further copied or distributed, except for government purposes, if the paper acknowledges U. S. Government sponsorship. References to this work should be either to the published version, if any, or in the form "private communication." For information about the ideas expressed herein, contact the author directly. For information about this series, contact Microsystems Technology Laboratories, Room 39-321, MIT, Cambridge, MA 02139; (617) 253-0292.

Irredundant Sequential Machines Via Optimal Logic Synthesis

Srinivas Devadas,* Hi-Keung Tony Ma,

A. Richard Newton and Alberto Sangiovanni-Vincentelli

Department of Electrical Engineering and Computer Sciences
University of California, Berkeley

Abstract

It is well known that optimal logic synthesis can ensure fully testable combinational logic designs. In this paper, we show that optimal sequential logic synthesis can produce irredundant, fully testable finite state machines. Test generation algorithms can be used to remove all the redundancies in sequential machines resulting in a fully testable design. However, this method may require exorbitant amounts of CPU time. The optimal synthesis procedure presented in this paper represents a more efficient approach to achieve 100% testability.

Synthesizing a sequential circuit from a State Transition Graph description involves the steps of state minimization, state assignment and logic optimization. Previous approaches to producing fully and easily testable sequential circuits have involved the use of extra logic and constraints on state assignment and logic optimization. In this paper, we show that 100% testability can be ensured without the addition of extra logic and without constraints on the state assignment and logic optimization. Unlike previous synthesis approaches to ensuring fully testable machines, there is no area/performance penalty associated with this approach. This technique can be used in conjunction with previous approaches to ensure that the synthesized machine is easily testable.

Given a State Transition Graph specification, a logic-level automaton that is fully testable for all single stuck-at faults in the combinational logic without access to the memory elements is synthesized. This procedure represents an alternative to a Scan Design methodology without the usual area and performance penalty associated with the latter method.

1 Introduction

Test generation for sequential circuits has long been recognized as a difficult task [5]. A popular approach to solving this problem is to make all the memory elements controllable and observable, i.e. Complete Scan Design [9] [1]. Scan Design approaches transform the sequential testing problem into one of combinational test generation which is considerably less difficult. They also remove all sequential redundancies in a circuit, since direct access is provided to the memory elements. However, there are situations where the cost in terms of area and performance of Complete Scan Design is not affordable. Also, the testing time associated with Scan Design is higher than that of a non-scan design, because values have to be sequentially scanned into and out of the memory elements one clock cycle at a time.

It is well known that optimal logic synthesis can ensure fully testable combinational logic designs. In this paper, we show that

optimal sequential logic synthesis can produce fully testable non-scan finite state machines. Test generation algorithms can be used to remove all the redundancies in sequential machines resulting in fully testable designs. However, in general, this method requires exorbitant amounts of CPU time. The optimal synthesis procedure presented in this paper represents a more efficient approach to achieve 100% testability.

Synthesizing a sequential circuit from a State Transition Graph description involves the steps of state minimization, state assignment and logic optimization. Previous approaches (e.g. [8]) to producing fully and easily testable sequential circuits have entailed the use of extra logic and constraints on state assignment and logic optimization. In this paper, we show that 100% testability can be ensured without the addition of extra logic and without constraints on the state assignment and logic optimization. This technique can be used in conjunction with previous approaches to ensure that the synthesized machine is easily testable.

The finite automaton is represented by a State Transition Graph, truth table or by an interconnection of gates and flip-flops. The synthesized/re-synthesized logic-level implementation is guaranteed to be fully testable for all single stuck-at faults in the combinational logic without access to the memory elements. This procedure represents an alternative to a Scan Design methodology without the usual area and performance penalty associated with the latter method.

Basic definitions and terminologies used are given in Section 2. Various types of redundant faults in sequential circuits are described in Section 3. In Section 4, we outline an optimal synthesis procedure of state minimization, state assignment and logic optimization that produces a highly testable Moore or Mealy finite state machine beginning from a State Transition Graph description. Any existing sequentially redundant faults in this machine are implicitly removed using extended don't care sets in repeated combinational logic minimization. These don't care sets are derived using techniques that check for state equivalence. We give theorems which prove the correctness of these procedures. In Section 5, we discuss the effects of redundancy removal on the state encoding of the machine. Preliminary results, which indicate that these procedures are viable for medium-sized circuits, are given in Section 6.

2 Preliminaries

A variable is a symbol representing a single coordinate of the Boolean space (e.g. a). A literal is a variable or its negation (e.g. a or \bar{a}). A cube is a set C of literals such that $x \in C$ implies $\bar{x} \notin C$ (e.g., $\{a, b, \bar{c}\}$ is a cube, and $\{a, \bar{a}\}$ is not a cube). A cube represents the conjunction of its literals. The trivial cubes, written 0 and 1, represent the Boolean functions 0 and 1 respectively.

*Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge

An expression is a set f of cubes. For example, $\{\{a\}, \{b, \bar{c}\}\}$ is an expression consisting of the two cubes $\{a\}$ and $\{b, \bar{c}\}$. An expression represents the disjunction of its cubes.

A cube may also be written as a bit vector on a set of variables with each bit position representing a distinct variable. The values taken by each bit can be 1, 0 or 2 (don't care), signifying the true form, negated form and non-existence respectively of the variable corresponding to that position. A minterm is a cube with only 0 and 1 entries.

A finite state machine is represented by its State Transition Graph (STG), $G(V, E, W(E))$ where V is the set of vertices corresponding to the set of states S , where $|S| = N_s$ is the cardinality of the set of states of the FSM, an edge joins v_i to v_j if there is a primary input that causes the FSM to evolve from state v_i to state v_j , and $W(E)$ is a set of labels attached to each edge, each label carrying the information of the value of the input that caused that transition and the values of the primary outputs corresponding to that transition. In general, the $W(E)$ labels are Boolean expressions. The number of inputs and outputs are denoted N_i and N_o respectively. The input combination and present state corresponding to an edge or set of edges is (i, s) , where i and s are cubes. The fanin of a state, q is a set of edges and is denoted $fanin(q)$. The fanout of a state q is denoted $fanout(q)$. The output and the fanout state of an edge $(i, s) \in E$ are $o((i, s))$ and $n((i, s)) \in V$ respectively.

Given N_i inputs to a machine, 2^{N_i} edges with minterm input labels fan out from each state. A STG where the next state and output labels for every possible transition from every state are defined corresponds to a completely specified machine. An incompletely specified machine is one where at least one transition edge from some state is not specified.

A starting or initial state is assumed to exist for a machine, also called the reset state. Given a logic-level finite state machine with N_b latches, 2^{N_b} possible states exist in the machine. A state which can be reached from the reset state via some input vector sequence is called a valid state in the STG. The input vector sequence is called the justification sequence for that state. A state for which no justification sequence exists is called an invalid state. Given a fault F , the State Transition Graph of the machine with the fault is denoted G^F . Two states in a State Transition Graph G are equivalent if all possible input sequences when the machine is initially in either of the two states produce the same output response.

A State Transition Graph G_1 is said to be isomorphic to another State Transition Graph G_2 if and only if they are identical except for a renaming of states.

The fault model assumed is single stuck-at. A finite state machine is assumed to be implemented by combinational logic and feedback registers. Tests are generated for stuck-at faults in the combinational logic part.

A primitive gate in a network is prime if none of its inputs can be removed without causing the resulting circuit to be functionally different. A gate is irredundant if its removal causes the resulting circuit to be functionally different. A gate-level circuit is said to be prime if all the gates are prime and irredundant if all the gates are irredundant. It can be shown that a gate-level circuit is prime and irredundant if and only if it is 100% testable for all single stuck-at faults.

We differentiate between two kinds of redundancies in a sequential circuit. If the effect of the fault cannot be observed at the primary outputs or the next state lines, beginning from any

state, with any input vector, the fault is deemed combinationally redundant. A sequentially redundant fault is a fault that cannot be detected by any input sequence and is not combinationally redundant.

To detect a fault in a sequential machine, the machine has to be placed in a state which can then excite and propagate the effect of the fault to the primary outputs. The first step of reaching the state in question is called state justification. The second step is called fault excitation-and-propagation.

An edge in a State Transition Graph of a machine is said to be corrupted by a fault if either the fanout state or output label of this edge is changed because of the existence of the fault. A path in a State Transition Graph is said to be corrupted if at least one edge in the path has been corrupted.

A multiple F-type fault for a line L , (which is the output of a gate and not a primary output), in a combinational network corresponds to a multiple fault condition on the fanout branches of line L . The multiple fault depends on the types of gates that L feeds into. For example, if a line L_1 has three fanout branches a , b , c , that feed into AND, OR, AND gates respectively, then the multiple F-type fault for L_1 is a stuck-at-1, b stuck-at-0 and c stuck-at-1. If the multiple F-type fault for a line is redundant, it means that the line (and all its fanout branches) can be bodily removed.

3 Origin of Redundant Faults in Sequential Circuits

There are two classes of redundant faults in a sequential circuit, namely, combinationally and sequentially redundant faults. Combinationally redundant faults (CRFs) are due to the presence of lines/wires in the logic circuit that do not contribute to the primary output or the next state functions. Replacement of these lines by constants will not change the functionality of the combinational logic in the sequential circuit. CRFs cannot be detected even if all the memory elements of the sequential circuit are made scannable. Sequentially redundant faults (SRFs), on the other hand, are related to the temporal characteristics of the sequential circuit. Although SRFs alter the combinational logic function of the circuit and hence the State Transition Graph (STG) representing the sequential circuit, they cannot be detected without making some of the latches scannable.

We now provide a definition of sequentially redundant faults.

1. An equivalent-SRF is a fault which causes only interchange and/or creation of equivalent states in the STG of the finite state machine.
2. An invalid-SRF does not corrupt any fanout edge of a valid state reachable from the reset state.
3. An isomorph-SRF transforms the original machine isomorphically, i.e. the faulty machine is equivalent to the good machine but with a different encoding. (There exists an isomorphism between the original and the faulty machine.)

We will use an example to illustrate the existence of sequentially redundant faults.

The State Transition Graph (STG) of a finite state machine is shown in Figure 1. The machine has 5 states and the states 010 and 110 are equivalent. The logic implementation of the combinational part of the machine is shown in Figure 2. The

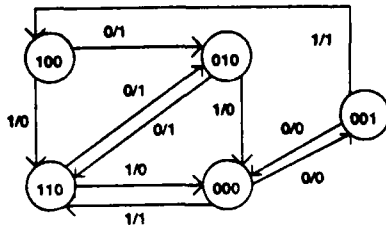


Figure 1: Original Finite State Machine

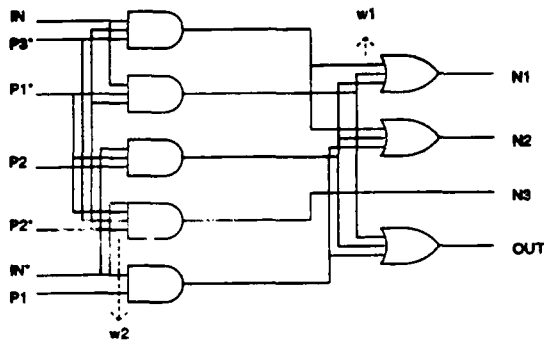


Figure 2: Combinational Logic of FSM

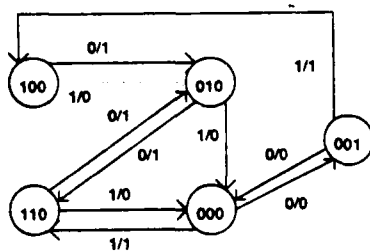


Figure 3: Faulty FSM with w1 s-a-0

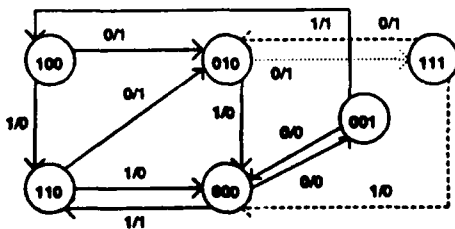


Figure 4: Faulty FSM with w2 s-a-1

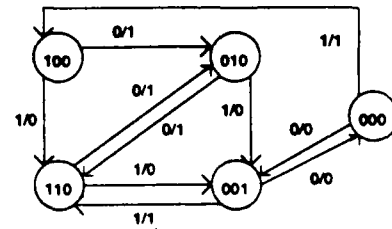


Figure 5: Faulty FSM with an isomorph-SRF

fault w1 stuck-at-0 (s-a-0) changes the original STG to the one shown in Figure 3. The corrupted edge is shown via a dotted line. Since 010 and 110 are equivalent states in the original STG, the fault w1 s-a-0 only causes an interchange of two equivalent states of the machine and is therefore sequentially redundant. The fault w2 s-a-1 changes the machine to the one shown in Figure 4. The fault creates an extra state 111, that was originally an invalid state which is equivalent to the true state 110. Therefore the fault w2 is also sequentially redundant. The corrupted edge is shown in dotted lines and the added edges shown in dashed lines.

If the detection of a fault in the combinational logic requires the machine to be brought to an invalid state (e.g. 101), then the fault is an invalid-SRF. An isomorph-SRF may change the original machine to the one shown in Figure 5. Note that the faulty machine represents an equivalent machine with a different encoding. The encodings for the states 000 and 001 in the original machine have been swapped. An isomorphism exists between the original and the faulty machine.

Theorem 3.1 : *A redundant fault in a finite state machine is either a CRF or an equivalent-SRF or an invalid-SRF or an isomorph-SRF.*

Proof (by contradiction): Assume a fault, F , is a redundant fault but not a CRF or equivalent-SRF or invalid-SRF or isomorph-SRF. Since F is not a CRF or an invalid-SRF, there must be an input sequence, beginning from the reset state, that will bring the machine to a state that can excite the fault and propagate its effect at least to some of the next state lines. Since F is not an equivalent-SRF or an isomorph-SRF, the fault effect on the next state lines will not cause an interchange or creation of equivalent states or an isomorphic mapping of states. This means the good state and the faulty state can be differentiated by a propagation sequence, i.e. the fault effect is propagated to the primary outputs, which means that the fault is testable. Q.E.D.

Theorem 3.1 guarantees that a fully testable finite state machine results if we ensure that none of these 4 kinds of redundancies described above exist in the synthesized machine. Steps in our synthesis procedure are designed to achieve this goal.

4 Irredundant Fully Testable Sequential Machines

A general model for a Mealy finite state machine is shown in Figure 6. It is realized by a combinational logic block, which implements the output and next state logic functions, and feedback registers. The Moore machine can be viewed as a special case of

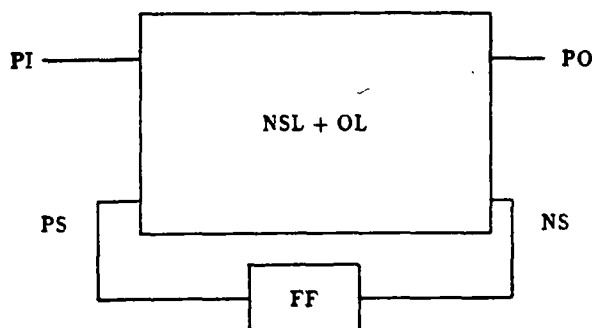


Figure 6: General Sequential Machine Model

a Mealy machine, where the outputs depend only on the present state of the machine.

We first describe the optimal synthesis procedure in Section 4.1. In Section 4.2, we prove that the resulting machine has no CRFs, invalid-SRFs or isomorph-SRFs. Experimental results indicate that the machine has very few redundancies. In Section 4.3, we present a modified synthesis procedure using extended don't care sets in repeated combinational logic minimization which ensures that equivalent-SRFs do not exist in the synthesized machine. The synthesized machine is thus made fully testable. In Section 4.4, we briefly discuss how finite automata represented at the truth table or at the logic-level can be made fully testable.

4.1 The Synthesis Procedure

The procedure consists of the steps of state minimization, state assignment and combinational logic optimization. These steps are described in the sequel.

1. **State Minimization:** Given an original State Transition Graph specification G^O we obtain a state minimum representation, G^M , using algorithms similar to those proposed in [14]. G^M has N_s valid states and satisfies the property that no two states are equivalent. State minimization for completely specified State Transition Graphs can be accomplished in $O(N \log N)$ time where N is the number of states in the machine, but is NP-complete for incompletely specified machines.
2. **State Assignment:** We encode the states in G^M , namely Q . The number of encoding bits N_b can be arbitrarily large ($N_b \geq \log_2(|Q|)$). State assignment algorithms like those in [13] and [7] can be used, which find a state assignment that heuristically minimizes the area of the combinational network after optimization. However, the state assignment algorithm may have to explore a certain number of possible state assignments in order to ensure a locally optimal solution (see Definition 4.2).
3. **Combinational Logic Optimization:** Given the encoded machine, which is now a combinational logic specification, we synthesize a prime and irredundant combinational logic network which implements both the next state logic and output logic functions. The transitions from the unused state codes,

are used as don't cares during the minimization. The number of inputs to the network will be $N_i + N_b$ and the number of outputs will be $N_o + N_b$. Prime and irredundant two-level networks can be produced using two-level logic minimizers like ESPRESSO [3]. Prime and irredundant multi-level networks can be synthesized using techniques like those in [2]. The multi-level network has to be irredundant for a certain class of multiple stuck-at faults as well (see Lemma 4.2).

We will have N_b latches in the synthesized sequential machine (denoted S^M) and 2^{N_b} valid and invalid states in the completely specified State Transition Graph (denoted G).

4.2 Correctness of Procedure

We can prove that the sequential machine synthesized by the procedure of the previous section is irredundant for all CRFs, invalid-SRFs and isomorph-SRFs.

The following theorem follows from the definition of state minimality. It is given in [11].

Theorem 4.1 : *Given a state minimized (reduced) machine M with N_s states, no machine with fewer states can realize the same terminal behavior. Also, any machine with the same number of states that realizes the same behavior has to be M or isomorphic to M .*

We now show that stuck-at faults cannot produce a faulty State Transition Graph that is isomorphic to the true State Transition Graph if the combinational logic implementing the next state and output logic functions is two-level, prime and irredundant. Isomorphic faulty and true State Transition Graphs imply that the fault has no other effect than interchanging the codes of the states of the machine.

The proof of this lemma can be found in Appendix A.

Lemma 4.1 : *Stuck-at faults on the primary input (PI), primary output (PO), present state (PS) and next state (NS) lines cannot produce a faulty State Transition Graph G^F that is isomorphic to G .*

Definition 4.1 : *A multi-level network is inversion-parity invariant if for any fault in the network, other than on the primary input lines, the parity of inversions is the same (either odd or even) for all paths to the primary outputs.*

Note that any two-level network is inversion-parity invariant. Also, networks that are synthesized by algebraic factorization from two-level networks are also inversion-parity invariant.

Theorem 4.2 : *If the two-level combinational circuit implementing the next state and output logic functions is prime and irredundant, then any fault F in the circuit cannot produce a G^F that is isomorphic to G . Also, if a prime and irredundant multi-level circuit is synthesized such that it is inversion-parity invariant, then any fault F in the circuit cannot produce a G^F that is isomorphic to G .*

Proof: By Lemma 4.1, we need not consider faults on the PI and PS lines. In a two-level network, faults on the intermediate lines and outputs, have the property that they either produce a D or a \bar{D} at the outputs of the network, uniformly for all test vectors that detect the fault. Isomorphism implies an interchange

of codes of multiple states. Without loss of generality, assume a two-way swap, between the codes of $q_1, q_2 \in G$ to produce G^F isomorphic to G . An edge e_1 exists from some state s_1 that goes to q_2 in G^F instead of q_1 in G . Similarly, an edge e_2 from some state s_2 , that goes to q_1 in G^F instead of q_2 in G exists. In the combinational sense, if e_1 produces a D at some next state line where q_1 and q_2 differ, e_2 has to produce a \bar{D} at that line. This is not possible in a two-level network for faults on intermediate lines and/or outputs. Therefore, isomorphism cannot occur.

The same argument holds for a inversion-parity invariant, prime and irredundant, multi-level network. **Q.E.D.**

In a general multi-level network, however, the faults in the intermediate lines may produce both a D as well as a \bar{D} at any particular output, due to reconvergent fanout paths with differing numbers of inversions. The arguments of Theorem 4.2 do not hold, when Boolean operations are used in multi-level combinational logic synthesis.

The proof of the following lemma can be found in Appendix B. A multi-level network can be made prime and irredundant for multiple stuck-at faults via the procedure of [2].

Lemma 4.2 : *If a prime and irredundant multi-level network C , with m outputs and asserting all 2^m output combinations, is irredundant for multiple F -type faults for each line in the network that is the output of a gate and not a primary output, then for any single stuck-at fault, F , in C , there will exist an input vector pair (i_1, i_2) such that i_1 is a test vector for the fault and i_2 is not, and i_1 produces the same output in C^F as i_2 does in C .*

Using Lemma 4.2, we can prove the following theorem, that restricts the occurrence of isomorphism in sequential machines, implemented by prime and irredundant multi-level networks that are also irredundant for multiple F -type faults in the network. Q denotes the set of states in G^M .

Theorem 4.3 : *If a set of states $Q_1 \in Q$ is such that each state in Q_1 has the property that its fanout edges assert distinct outputs from all other states in Q or has fanout next states in $Q - Q_1$, which are distinct from the fanout states of all other states in Q , or possesses distinct combinations of outputs and fanout next states, then a fault cannot produce an isomorphic machine causing only interchange of states within Q_1 .*

Proof: We will first prove the case of $||Q_1|| = 2$ and where fanout edges from state s_1 assert a set of distinct outputs O_1 and fanout edges from the second state s_2 assert a set of distinct outputs O_2 . Assume there exists a fault F that produces an isomorphism between these states. In the isomorph G^F , fanout edges from s_1 (s_2) will assert O_2 (O_1). However, by Lemma 4.2, an uncorrupted edge asserting some $o \in O_1$ or $o \in O_2$ has to exist in G^F . This edge can only come from s_1 or s_2 , respectively. This means that in the faulty machine, either s_1 or s_2 asserts outputs from both O_1 and O_2 , implying that G^F is not isomorphic to G . The argument is easily generalized to $||Q_1|| > 2$.

A similar argument can be made for states s_1, s_2 with distinct next state fanouts or distinct combinations of outputs and next state fanouts. **Q.E.D.**

Thus, a sequential machine with a G^M where all states possess distinct combinations of outputs and fanout states cannot have faults that cause isomorphism, whether the combinational logic is implemented in two-level or general multi-level form.

Definition 4.2 : *A state assignment of G^M is deemed to be locally optimal with respect to a subset of states $Q_1 \in G^M$, if interchanging the codes of $q \in Q_1$ does not produce, after optimization, a logic implementation that is exactly the same as the previous one, except with one less literal.*

The state assignment is locally rather than globally optimal in the sense that interchanging the code of $q_1 \in Q_1$ with $q_2 \notin Q_1$ could produce a better logic implementation. In a multi-level implementation, if there exist states in G^M that do not satisfy the condition of Theorem 4.3, then in order to ensure that a redundant fault does not cause isomorphism, the state assignment of G^M has to be locally optimal, with respect to interchanging the codes of these states. For a two-level implementation, any state assignment is locally optimal, with respect to all states in G^M .

Theorem 4.4 : *If G^M contains 2^{N_b} valid states where N_b is the number of latches in S^M , S^M is fully testable, if the prime and irredundant combinational network is implemented in two-level form, or if a locally optimal state assignment has been found, as per Definition 4.2, across all states that do not satisfy the condition of Theorem 4.3.*

Proof: No fault in the machine can result in an increase in the number of states, since the true machine has the maximum possible number of states, namely 2^{N_b} . Since G^M is reduced, we know that no machine with fewer than 2^{N_b} states can realize the behavior of G^M . All faults are combinational irredundant, since the combinational logic is prime and irredundant. For a combinational irredundant fault F to be sequentially redundant, the faulty machine G^F has to be isomorphic to the true machine G . By Theorem 4.2 this is not possible in a two-level implementation. In a multi-level implementation, if G^F is isomorphic to G , the sets of states satisfying the condition of Theorem 4.3 cannot be involved in the isomorphism. If isomorphism occurs due to F , it has to involve a set of states, Q_1 , not satisfying the condition of Theorem 4.3. The isomorphism produces a G^F equivalent to G , with a better implementation (after optimization) than that of G (with at least one less line). However, this contradicts the fact that the initial state assignment for G^M that produced G is locally optimal under the exchange(s) of the codes of states in Q_1 . Therefore, S^M is fully testable. **Q.E.D.**

The above theorem is quite a strong result. Given a State Transition Graph G^M , if extra states can be added to G^M such that the resulting graph $G^{M'}$ is reduced and has 2^n states, then the synthesized machine $S^{M'}$ is guaranteed to be fully testable, provided the state assignment is locally optimal. Of course, adding the extra states and edges to G^M constitutes an area overhead. If G^M has less than 2^{N_b} states, the unused state codes can be used as don't care states to minimize the combinational specification.

The proof of this lemma can be found in Appendix C.

Lemma 4.3 : *An invalid state in the State Transition Graph is never required to detect a fault in S^M .*

We now use the preceding results to prove the partial irredundancy theorem for machines where G^M has $N_s < 2^{N_b}$ states.

Theorem 4.5 : *The sequential machine S^M produced by the synthesis procedure may contain only equivalent-SRFs.*

Proof: By Lemma 4.3, no invalid-SRFs can exist. By Theorem 4.2, if S^M is implemented as a two-level network, no isomorph-SRFs can exist. If S^M is implemented as a multi-level network, then a locally optimal state assignment as per Definition 4.2, across all states that do not satisfy the condition of Theorem 4.3, is found. This guarantees that no isomorph-SRFs will exist. S^M does not contain any CRFs. Therefore, by Theorem 3.1, only equivalent-SRFs can exist. **Q.E.D.**

4.3 Eliminating Redundancies Via Extended Don't Care Sets

In this section, we show how the testability of the synthesized machine S^M can be increased by removing possible equivalent-SRFs through succeeding logic minimization steps, *without explicitly identifying these redundancies*. Redundancies are identified and removed implicitly via the use of *extended don't care sets*.

A *simple* equivalent-SRF was illustrated in Figure 4 (Section 3). We have a situation where an invalid state q has identical faout and hence is equivalent to some valid state r_1 . An edge from v_2 to r_1 is corrupted to go to q . F only corrupts one edge in the State Transition Graph and propagates only one time-frame. In the general case, a equivalent-SRF can propagate multiple time-frames, when the invalid state q is equivalent to the true valid state r_1 , but does not have identical faout.

These redundancies are likely to occur, especially if a large number of unused state codes exist. These redundancies occur because current state assignment algorithms do not use the freedom of state splitting (Section 5), so as to obtain an optimal solution. It is very difficult to extend state assignment algorithms in this direction and hence we ensure irredundancy by specifying an extended don't care set in a repeated logic minimization procedure.

1. State assignment and logic optimization are performed as before, with logic optimization using the invalid states as don't cares.
2. Given the prime and irredundant logic network, the State Transition Graph, G , corresponding to the network is extracted. All invalid states $iv \in G$ that are equivalent to valid states $r \in G$ are found. It should be noted that G is a completely specified combinational logic function, corresponding to an encoded State Transition Graph.
3. Given a valid state r_1 , valid states r_2, r_3, \dots, r_L that are equivalent to r_1 and invalid states iv_1, iv_2, \dots, iv_K that are equivalent to r_1 , then the faout of r_1 is re-specified as $n(fanin(r_1)) = DC(r_1, r_2, \dots, r_L, iv_1, iv_2, \dots, iv_K)$. $DC()$ implies that any (but at least one) of the enclosed state entries can be used. In practice, if r_1 and some or all of the $iv_k, 1 \leq k \leq K$ can be merged into a single cube, c , then every occurrence of v_1 in the next state field of G is replaced by c .¹ G with this extended don't care set is made prime and irredundant via logic minimization to produce G' . This may make a previously invalid state valid.
4. G' may have some invalid states, which could be different from the invalid states in G . These invalid state codes are

¹If the codes cannot be merged into a single cube, we have a Boolean relation (4) corresponding to the permissible next states of the edge and the combinational logic has to be optimized with respect to this Boolean relation.

used as don't cares and G' is made prime and irredundant under this new don't care set to produce G'' .

5. If $G' = G''$, exit. Else $G \leftarrow G''$, go to Step 2.

In the first iteration, there will not be valid states r_2, \dots, r_L that are equivalent to any r_1 , since we begin with a reduced machine. However, after Step 3 above, some invalid states that are equivalent to r_1 may become valid.

Theorem 4.6 : *The procedure above converges, and the resulting machine after convergence will not have any simple equivalent-SRFs, invalid-SRFs or isomorph-SRFs.*

Proof: The procedure converges when succeeding logic minimizations have produced the same result. Each logic minimization starts with the result of the previous logic minimization. Additional don't cares are provided. We are guaranteed that the overall cost function (e.g. the number of lines in the network) has a finite decrease if the logic function is altered. Since the cost function is bounded from below, the sequence of logic minimizations must eventually converge, and on the last call, return an unchanged network, η . No isomorph-SRFs will exist in the prime and irredundant network η by Theorem 4.2 and Theorem 4.3. Since the invalid states have been used as don't cares to produce η and the network is unchanged since then (even though additional minimizations may have been performed), no invalid-SRFs can exist.

Finally, using the don't care sets corresponding to the equivalent states, ensures that for each fault F there will exist at least one corrupted edge that goes to a state, q^F , that is *not* equivalent to the true next state, q , in the true machine G , regardless of whether the q^F is invalid or valid. η is unchanged since the use of the invalid states as don't cares, so an edge fanning out of a valid state has to exist with this property. $q^F \in G^F$ has to become equivalent to $q \in G$ for F to be redundant, but that would mean that F is not a simple equivalent-SRF. Therefore, F is testable or not a simple equivalent-SRF. **Q.E.D.**

More complicated equivalent-SRFs may exist, though experimental evidence indicates that this is extremely rare. In fact, we have yet to encounter a *single* case of an equivalent-SRF that is not of the form of the SRF of Figure 4. These redundancies correspond to the case, where $q^F \in G$ is not equivalent to $q \in G$ but $q^F \in G^F$ becomes equivalent to $q \in G$, making F redundant. A larger set of don't cares can ensure that these equivalent-SRFs do not occur in the machine. The synthesis procedure described above is unchanged except for introducing an additional don't care set in Step 3 where G' is produced, as described below.

Step 3b: Given a state q_2 that is not equivalent to a valid state q_1 , the set of input combinations $i_{nc}(q_1, q_2)$ are found which make this pair not equivalent. If q_2 were equivalent to q_1 then $i_{nc} = 0$. The don't care specification is $n(fanin(q_1)) = DC(q_1, q_2)$, with a constraint on a subset of faout edges of q_2 if q_2 is picked rather than q_1 . The constraint for a single cycle propagation is that

$$\alpha(i_{nc}(q_1, q_2), q_2) = \alpha(i_{nc}(q_1, q_2), q_1) \bigwedge n(i_{nc}(q_1, q_2), q_2) = n(i_{nc}(q_1, q_2), q_1)$$

This set of don't cares and associated constraints are found for the different state pairs that are not equivalent. Optimal use of these don't cares and associated constraints, generalized to multiple-cycle propagation, ensures full testability.

Theorem 4.7 : *Using the additional don't care set in the synthesis procedure will result in a fully testable machine.*

Proof: By Theorem 4.6, no simple equivalent-SRFs, invalid-SRFs or isomorph-SRFs will exist in the machine. Using the additional don't cares will ensure that there will always be an edge from a valid state that is corrupted to q^F instead of q such that $q^F \in G \neq q \in G$ and $q^F \in G^F \neq q \in G$. Therefore, G^F and G can be differentiated by distinguishing q^F and q and F is testable. Q.E.D.

The enhanced procedure will remove all equivalent-SRFs in the machine which has been synthesized as described in the previous section. In practice, *only the simple don't cares of Step 3 suffice to ensure full testability*, allowing a locally optimal solution with no redundancies to be reached: the more complicated don't cares of Step 3b are *not* required. That is fortunate, since current logic optimization programs are quite restricted in the specification and optimal usage of don't cares.

The procedure is quite CPU-intensive since repeated combinational logic minimizations have to be performed. Experimental results (Section 6) indicate that the machine prior to using the extended don't care sets is highly testable, and in some cases, fully testable. Removing the few redundancies can be accomplished using reasonable amounts of CPU time. The fact that a network has to repeatedly be made prime and irredundant in order to ensure full testability for a sequential circuit, indicates that synthesizing irredundant sequential circuits is more difficult than synthesizing irredundant combinational circuits.

4.4 Synthesis from Logic-Level Descriptions

In this section, we describe how complete or partial re-synthesis of logic-level circuits can be performed so as to ensure irredundant sequential machines. Given a combinational specification of a circuit in the form of a truth table, i.e. a previously encoded finite state machine, the following steps are performed in re-synthesis. The combinational specification has $N_i + N_b$ inputs and $N_o + N_b$ outputs, where N_b is the number of encoding bits used (latches) in the state assignment process.

1. The combinational specification is made disjoint in the present state field (the last N_b inputs). A cube entry in the field is identical to another cube entry or does not intersect it. A two-level cover can be made disjoint using the disjoint SHARP operation in [3].
2. The specification is now treated as a State Transition Table, with each distinct entry in the present state and next state field representing a distinct state. If some states cannot be reached from the reset state (invalid states), they are deleted from the description. The State Table is now state minimized. Some states (represented by cubes or minterms) may be removed because of being equivalent to other states.
3. The encoded State Transition Table represents a combinational logic specification that can be made prime and irredundant. A fully testable machine can be synthesized via the procedures of Section 4.2 and 4.3.

The re-synthesis procedure can be extended to begin from a logic-level description. In this case, the State Transition Graph of the machine is extracted using the efficient cube-enumeration techniques presented in [6]. Given this (encoded) State Transition Graph, Steps 1-3 described above are carried out as before.

5 Effect of Redundancy Removal via Logic Minimization on State Encoding

If a combinational redundant line is removed from a logic network (i.e. replaced with a 0 or a 1), network functionality remains unchanged. Similarly, when a sequentially redundant but combinational irredundant line is removed from a sequential machine, the terminal behavior of the machine remains unchanged. However, the State Transition Graph of the machine, and the state encoding are affected by redundancy removal via repeated logic minimization.

Two things may happen during redundancy removal:

1. A state may be added to the State Transition Graph, which is equivalent to some other valid state. An edge is redirected from some valid state to this originally invalid state.
2. A valid state may be replaced by an originally invalid state. In effect, the encoding of a symbolic state is changed.

The occurrence of the first effect is due to the fact that state assignment is performed on a state minimized Graph. It is well known [10] that state splitting may be required for an optimal state assignment. Unfortunately, the state assignment problem is difficult enough, without adding the extra degree of freedom of being able to split states. The faulty, but equivalent, State Graph corresponds to a "better" state assignment with (at least) one state split into two (or more) components.

The occurrence of the second effect is due to a state assignment that is not locally optimal for the reduced State Graph, even without the addition of extra states. As mentioned in Section 4.2, when a machine has a two-level combinational logic implementation, any state assignment is locally optimal with respect to all the used state codes. However, the state assignment may be sub-optimal when considering the invalid or unused state codes. In the multi-level case too, a state assignment that is locally optimal under the valid (used) state codes may be sub-optimal when considering the invalid (unused) state codes. The replacement of a state code by an unused state code results in a "better" machine.

State assignment techniques (e.g. [7] [13]) do not take state splitting into account in their attempt to find locally or globally optimal solutions. In our experience, the occurrence of the first effect is much more frequent. If an optimal state assignment can be found exploiting the freedom of state splitting, then the resulting logic implementation will be fully testable. Repeated logic minimization, as described in Section 4.3, has the effect of changing a sub-optimal state encoding to a locally optimal encoding that corresponds to a fully testable machine.

6 Results

In this section, we present some preliminary results obtained using the synthesis procedures described in Section 4. Intensive optimization is necessary to obtain fully testable designs. If this optimization can be carried out, then the synthesized machine will occupy minimal area. There is no area/performance overhead associated with this procedure. However, the CPU time requirements have to be evaluated.

Redundancies can be explicitly removed via the use of test pattern generation algorithms, to produce fully testable sequential circuits. However, redundant lines corresponding to redundant stuck-at faults can only be removed (replaced with a 0 or a 1)

EX	#inp	#out	#states	#edges
ex1	2	2	6	24
ex2	2	1	13	57
bbara	4	2	7	45
bbsse	7	7	13	55
s1	8	6	20	110
planet	7	19	48	118
dfile	2	1	24	96
styr	9	10	30	165
keyb	7	2	19	170
scf	27	54	128	168

Table 1: Statistics of Benchmark Examples

one at a time. Furthermore, removing a redundant line may introduce new redundancies and so all faults have to be checked for redundancy on each removal. We compare these two techniques to the synthesis of irredundant sequential circuits.

We chose some examples in the MCNC 1987 Logic Synthesis Workshop as test cases, whose statistics are given in Table 1. Beginning from a State Transition Graph description, G , the following steps were performed in the synthesis procedure.

1. **State Minimization:** The machines were state minimized.
2. **State Assignment:** Binary codes were assigned to the states in G using the program KISS [13]. The encoding length in some cases was greater than the minimum required. The codes were all minterms, and some minterms were not used. The combinational logic specification, a truth table, after encoding is denoted T .
3. **Logic Optimization:** T , with all the unused state codes specified as don't cares, was optimized using ESPRESSO, and algebraically factored to produce a multi-level logic network C . C was prime and irredundant.

Tests were generated for the resulting sequential machine M whose combinational logic is implemented by C . Test generation was accomplished using the program STALLION [12]. The number of encoding bits used in state assignment (#lat), the number of gates in C (#gate) and the fault coverage obtained (fault cov.) by STALLION are given in Table 2. The CPU times for logic optimization (l.o. time), test generation (TPG time) and the number of test sequences (test seq.) generated are also given. All the undetected faults were checked for redundancy using algorithms in STALLION. The number of redundant faults (%red. fault) and the CPU time expended during redundancy identification (r.i. time) and redundancy removal (r.r. time) are given in Table 2. The CPU times for state assignment and the initial state minimization were negligible and are not given. In the tables, s stands for CPU seconds on a VAX 11/8650 and m for CPU minutes. For all the cases, the machine produced is highly testable. The larger examples, scf and planet which have significantly more outputs than latches are fully testable.

The redundancy identification times in Table 2 represent the CPU times required to explicitly identify redundant lines in the given circuit. Explicitly removing these redundancies in order to obtain a fully testable circuits requires considerably more CPU time as indicated in Table 2 (r.r. time). This method is only feasible for small examples.

EX	#lat.	#gate	fault cov.	l.o. time	TPG time	test seq.	%red. fault	r.i. time	r.r. time
ex1	3	23	97.92	0.5s	2.0s	19	2.08	1.1s	2.0s
ex2	5	35	98.15	2.2s	41.8s	22	1.85	6.1s	1.1m
bbara	3	56	100.0	1.2s	104.8s	42	0.0	0.0s	0.0s
bbsse	4	91	100.0	2.1s	3.2m	46	0.0	0.0s	0.0s
s1	5	105	99.79	5.5s	303s	74	0.21	4.0s	303s
planet	6	193	100.0	10.5s	141.8s	80	0.0	0.0s	0.0s
dfile	6	77	97.80	6.2s	331.8s	62	2.20	41.8s	> 1h
styr	5	367	100.0	80.4s	42.1m	165	0.0	0.0s	0.0s
keyb	5	146	98.65	29.5s	21.2m	101	1.35	1.2m	> 1h
scf	8	402	100.0	121.4s	82.2m	136	0.0	0.0s	0.0s

Table 2: Synthesis Procedure Results

EX	s.e. time	#logic mini.	l. o. time	fault cov.	TPG time
ex1	0.5s	1	0.5s	100.0	2.1s
ex2	6.5s	7	22.4s	100.0	40.6s
s1	1.0s	1	6.1s	100.0	298.2s
dfile	10.2s	3	25.5s	100.0	747.7s
keyb	14.6s	2	27.8s	100.0	21.6m

Table 3: Results using Extended Don't Care Sets in Synthesis

The number of test sequences generated for each example is comparable to the number of single test vectors generated via a Complete Scan Design approach. However, each test sequence has multiple test vectors (between 1-10) that have to be applied to the PI lines. In the Scan Design case, each test vector requires multiple clock cycles to be applied.

The examples of Table 2 with < 100% fault coverage were re-synthesized using the extended don't care set as described in Section 4.3. The CPU time to check for equivalence between invalid and valid states (s.e. time), number of logic minimizations (#logic mini.), CPU time spent in logic minimization (l.o. time), the final fault coverage (fault cov.) using STALLION and the test generation time (TPG time) are indicated in Table 3. The CPU time required for the state equivalence checks and the extra logic minimization steps are less than sequential test generation and redundancy removal times (Table 2), indicating that the optimal synthesis procedure is more efficient than an explicit redundancy identification method. Using the simple don't cares (Step 3 in Section 4.3) resulted in fully testable designs in all cases. We have yet to find an example where this is not the case.

7 Conclusions

We have described a synthesis procedure that produces an optimized, fully testable logic implementation of a sequential machine from a State Transition Graph description of the machine. During synthesis, possible redundancies in the machine are implicitly eliminated using state equivalence checking and combinational logic minimization. No direct access to the memory elements is required.

The optimal synthesis procedure described involves the steps

of state minimization, state assignment and logic optimization. It is applicable to Moore or Mealy finite state machines. This procedure has no associated area/performance overhead unlike Scan Design methodologies. It can be used in conjunction with previous synthesis approaches to ensure easily testable sequential machines. In this case, test sequences which detect all single stuck-at faults in the sequential machine can be obtained via combinational test generation and depth-first search on the State Transition Graph.

Ongoing work includes the generalization of these methods to arbitrary interconnections of finite state machines.

8 Acknowledgements

The interesting discussions with Kurt Keutzer and Robert Brayton on sequential circuit optimization and testability are acknowledged. This work was supported in part by the Semiconductor Research Corporation, the Defense Advanced Research Projects Agency under contract N00014-87-K-0825 and a grant from AT&T Bell Laboratories.

APPENDIX

A Proof of Lemma 4.1

Proof: Consider a primary input fault F . Without loss of generality, assume that it is a stuck-at-1 fault on the 1st primary input line. The effect of this fault is to cause all input vectors i_k such that $i_k[1] = 0$ to become, in effect, i_l where $i_l[1] = 1$ & $i_l[i] = i_k[i]$, $2 \leq i \leq N_j$. Since F is combinational irredundant, there will exist an input vector pair (i_1, i_2) where $i_1[1] = 0$, $i_2[1] = 1$ & $i_1[i] = i_2[i]$, $2 \leq i \leq N_j$ such that $n(i_1, q) \neq n(i_2, q) \parallel o(i_1, q) \neq o(i_2, q)$ for some q (Else, i_1 can be replaced by $i_1 \cup i_2$ in the combinational truth table). First, consider the case where the fanout states are different for i_1 and i_2 . If in G , $n(i_1, q) = q_2$ and $n(i_2, q) = q_3$, then in G^F we have $n(i_1, q) = n(i_2, q) = q_3$. For G^F to be equivalent to G , we need $q_3 \in G^F \equiv q_2 \in G$ and $q_3 \in G^F \equiv q_3 \in G$ (since there is a corrupted and uncorrupted edge from q to q_3 in G^F). This requires $q_3 \in G \equiv q_2 \in G$, which is a contradiction. The second case where the primary outputs of i_1 and i_2 are different is simpler. We have two edges from a state in G that assert different outputs and go to the same next state, merging in G^F . This means G^F cannot be isomorphic to G .

A primary output o exists in G^M , if and only if there exists a pair of edges e_1 and e_2 which assert both values of the output, 0/1. When the machine makes the transition corresponding to the edge which asserts the value of the output different from the stuck value, the fault will be detected.

If all stuck-at faults on present state lines are combinational irredundant, for any present state line i , there are two states q_1 and q_2 whose codes differ in bit i alone. q_2 and q_1 merge in G^F due to a fault on present state line i . Hence, $\|G^F\| < \|G\|$ and isomorphism cannot occur.

The argument for the next state line faults is similar to the argument for the present state line faults. **Q.E.D.**

B Proof of Lemma 4.2

Proof: Consider a prime and irredundant multi-level circuit im-

plementing G . The circuit is leveled from the primary outputs to the primary inputs. Gates generating primary outputs are assigned level 0 and a gate that drives gates with levels l_1, l_2, \dots, l_n has a level equal to $\max(l_i) + 1$. The gates at level j are $g_{j1}, g_{j2}, \dots, g_{jN_j}$. The outputs of these gates constitute a set of N_j variables $\Pi(j)(i)$, $1 \leq i \leq N_j$. The combinations of $\Pi(j)$ that are caused by some primary input combination are denoted $\Pi(j)^{CA}$ and the combinations that never appear are denoted $\Pi(j)^{DC}$.

Without loss of generality, consider the s-a-0 and s-a-1 faults on $\Pi(1)(1)$. Some $ir_1 \in \Pi(1)^{CA}$ has to detect the s-a-0 fault and some $ir_2 \in \Pi(1)^{CA}$ has to detect the s-a-1 fault. Obviously, $ir_1[1] = 1$ and $ir_2[1] = 0$. If for any $ir_1 \in \Pi(1)^{CA}$ that detects the s-a-0 fault, there is a $ir_3 \in \Pi(1)^{CA}$ such that $ir_3[1] = 0$, $ir_3[i] = ir_1[i]$, $2 \leq i \leq N_j$, then we have a complementary PI vector pair (ir_1, ir_3) corresponding to (ir_1, ir_3) with ir_1 detecting the s-a-0 fault and producing a faulty output equal to the true output of ir_3 which does not detect the fault. Furthermore, (ir_3, ir_1) will be a complementary PI vector pair for the s-a-1 fault.

We then consider the case of $ir_3 \in \Pi(1)^{DC}$ for all $ir_1 \in \Pi(1)^{CA}$ that detect the s-a-0 fault. By the argument above, if for any $ir_2 \in \Pi(1)^{CA}$ that detects the s-a-1 fault, there is a $ir_4 \in \Pi(1)^{CA}$ such that $ir_4[1] = 0$, $ir_4[i] = ir_2[i]$, $2 \leq i \leq N_j$, then (ir_2, ir_4) constitutes a complementary pair for the s-a-1 fault and (ir_4, ir_2) constitutes a complementary pair for the s-a-0 fault.

The last case we need to consider is $ir_3 \in \Pi(1)^{DC}$ for all $ir_1 \in \Pi(1)^{CA}$ that detect the s-a-0 fault and $ir_4 \in \Pi(1)^{DC}$ for all $ir_2 \in \Pi(1)^{CA}$ that detect the s-a-1 fault on $\Pi(1)(1)$. For any $ir_k \in \Pi(1)^{CA}$ that does not detect the s-a-0 or s-a-1 fault, we have ir_l such that $ir_l[1] = \bar{ir}_k[1]$, $ir_l[i] = ir_k[i]$, $2 \leq i \leq N_j$, producing the same output as ir_k in the true or faulty circuit. We then can represent $\Pi(1)^{CA}$ using $\Pi(1)^{DC}$ as a set of cubes, $ir_1 \cup ir_3, ir_2 \cup ir_4, \dots, ir_k \cup ir_l$, where the first bit in each cube is a don't care. This means the line $\Pi(1)(1)$ can be bodily removed, i.e. the multiple F-type fault corresponding to $\Pi(1)(1)$ is redundant, which is a contradiction. Therefore, a complementary vector pair has to exist for the stuck-at faults on $\Pi(1)(1)$ and other $\Pi(1)(k)$.

A similar argument can be made for the intermediate lines corresponding to the inputs to the g_{ji} , using the fact that the m -output, fault-free network asserts all distinct 2^m output combinations. **Q.E.D.**

C Proof of Lemma 4.3

Proof: All unused state codes may be used as don't cares during logic minimization. Invalid states can only correspond to some unused state code. Since the combinational network is prime and irredundant under this don't care set, there always exists a valid state that detects any fault (and provides the initial propagation to the next state lines or primary outputs) that the invalid state detects. **Q.E.D.**

References

- [1] V. D. Agarwal, S. K. Jain, and D. M. Singer. Automation in design for testability. In *Proc. of Custom Integrated Circuit Conference*, May 1984.

- [2] K. Bartlett, R. K. Brayton, G. D. Hachtel, R. M. Jacoby, C. R. Morrison, R. L. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang. Multi-level Logic Minimization Using Implicit Don't Cares. In *IEEE Transactions on Computer-Aided Design*, pages 723-740, June 1988.
- [3] R. K. Brayton, G. D. Hachtel, Curt McMullen, and A. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, 1984.
- [4] R. K. Brayton and F. Somenzi. Boolean Relations and the Incomplete Specification of Logic Networks. In *Proc. of VLSI 89*, August 1989.
- [5] M. A. Breuer and A. D. Friedman. *Diagnosis and Reliable Design of Digital Systems*. Computer Science Press, 1976.
- [6] S. Devadas, H-K. T. Ma, and A. R. Newton. On the verification of sequential machines at differing levels of abstraction. In *IEEE Transactions on Computer-Aided Design*, pages 713-722, June 1988.
- [7] S. Devadas, H-K. T. Ma, A. R. Newton, and A. Sangiovanni-Vincentelli. Mustang: state assignment of finite state machines targeting multi-level logic implementations. In *IEEE Transactions on Computer-Aided Design*, pages 1290-1300 December 1988.
- [8] S. Devadas, H-K. T. Ma, A. R. Newton, and A. Sangiovanni-Vincentelli. A Synthesis and Optimization Procedure for Fully and Easily Testable Sequential Machines. In *IEEE Transactions on Computer-Aided Design*, October 1989. to appear.
- [9] E. B. Eichelberger and T. W. Williams. A Logic Design Structure for LSI Testability. In *Proc. 14th Design Automation Conference*, pages 462-468, June 1977.
- [10] J. Hartmanis and R. E. Stearns. Some dangers in the state reduction of sequential machines. In *Information and Control*, pages 252-260, September 1962.
- [11] F. J. Hill and G. R. Peterson. *Introduction to Switching Theory and Logical Design*. John Wiley and Sons, 1981.
- [12] H-K. T. Ma, S. Devadas, A. R. Newton, and A. Sangiovanni-Vincentelli. Test generation for sequential circuits. In *IEEE Transactions on Computer-Aided Design*, pages 1081-1093, October 1988.
- [13] G. De Micheli, R. K. Brayton, and A. Sangiovanni-Vincentelli. Optimal State assignment of Finite State Machines. In *IEEE Transactions on Computer-Aided Design*, pages 269-285, July 1985.
- [14] M. C. Paull and S. H. Unger. Minimizing the number of states in incompletely specified sequential circuits. In *IRE Transactions on Electronic Computers*, pages 356-357, September 1959.